

# C# Avançado

## LINQ: Language Integrated Query



---

---

---

---


---

---

---

---

Tópicos Abordados



- Conceitos de C# usados pelo LINQ
- Funcionamento do LINQ
- Deferred e immediate execution
- A classe *System.Linq.Enumerable*
- Método *OfType<T>()*
- Projetando tipos de dados anônimos
- Operações com conjuntos
- Agrupamento de dados
- Operador *join*
- Geração de dados

---

---

---

---


---

---

---

---

LINQ



- Permite a construção de queries para extrair dados
  - O LINQ tem algumas similaridades com a linguagem SQL
- Extração de dados de várias fontes
  - Objetos (coleções e arrays)
  - XML
  - *DataSet*
  - Entities

---

---

---

---

---

---

---

---

## Conceitos Necessários



- O uso de LINQ se apoia em alguns conceitos da linguagem C#
  - Expressões lambda
  - Extension methods
  - Tipo de dados *var*
  - Tipos anônimos
- Os dois últimos serão abordados a partir de agora

---

---

---

---

---

---

---

## O Tipo *var*



- Variáveis locais em C# podem ser declaradas usando o tipo *var*
  - O tipo de dado da variável é definido de forma implícita pelo compilador

```
var i = 10;           // tipo 'int'
var s = "ABC";        // tipo 'string'
var b = true;         // tipo 'bool'
var p = new Pessoa(); // tipo 'Pessoa'
```

- Depois que o tipo da variável é definido, não pode ser trocado

```
var v = 10;
v = "ABC";
```

Não compila

---

---

---

---

---

---

---

## Regras de Uso do *var*



- Uso restrito a variáveis locais (definidas dentro de método ou properties)
  - Parâmetros de métodos, retorno ou fields não podem usar o *var*
- A variável deve receber um valor no momento da declaração
  - É através do valor que o compilador vai decidir o tipo de dado de forma implícita
  - A primeira atribuição não pode ser *null*

---

---

---

---

---

---

---

## Tipos Anônimos



- Um tipo anônimo é uma classe criada sem nome
  - O nome é gerado apenas no momento da compilação
  - O nome não precisa ser conhecido
- Tipos anônimos são bastante úteis em situações onde você deseja criar classes que existem apenas para encapsular dados e serão usadas apenas em locais específicos do código

---

---

---

---

---

---

---

## Definindo Tipos Anônimos



- A variável que vai referenciar o objeto do tipo anônimo deve ser do tipo *var*
  - O tipo não é conhecido durante a escrita do código. Só será conhecido na compilação

```
var cachorro = new { Nome = "Totó", Idade = 5 };
```

Variável que referencia um objeto da classe anônima

---

---

---

---

---

---

---

## O Tipo Anônimo Internamente



- Como a classe é definida internamente?

```
internal sealed class AnomClass
{
    private readonly string nome;
    private readonly int idade;

    public string Nome { get { return nome; } }
    public int Idade { get { return idade; } }

    public AnomClass(string nome, int idade)
    {
        this.nome = nome;
        this.idade = idade;
    }
    ...
}
```

O nome da classe é interno

---

---

---


---

---

---

---

O Tipo Anônimo Internamente



```

public override string ToString()
{
    return new StringBuilder()
        .Append(" Nome = ").Append(nome)
        .Append(" Idade = ").Append(idade)
        .Append(" ")
        .ToString();
}

public override bool Equals(object obj)
{
    AnomClass other = obj as AnomClass;
    if (other == null)
        return false;

    return this.nome == other.nome && this.idade == other.idade;
}

public override int GetHashCode()
{
    StringBuilder builder = new StringBuilder();
    builder.Append(nome).Append(idade);
    return builder.ToString().GetHashCode();
}

```

---

---

---

---


---

---

---

---

Múltiplas Instâncias de Tipos Anônimos



- Se tipos anônimos forem definidos mais de uma vez usando as mesmas propriedades, o compilador gera apenas uma classe

```

var cachorro1 = new { Nome = "Totó", Idade = 5 };
var cachorro2 = new { Nome = "Rex", Idade = 8 };

```

Mesma classe

```

var cachorro1 = new { Nome = "Totó", Idade = 5 };
var cachorro2 = new { Nome = "Rex", Peso = 3.5 };

```

Classes diferentes

Ao comparar objetos, use o método Equals()

---

---

---

---

---

---

---

---

Visão Geral do LINQ



- O LINQ trabalha com extração de dados



---

---

---

---


---

---

---

---

## Exemplo de Uso do LINQ



```
List<Pessoa> pessoas = new List<Pessoa> {
    new Pessoa { Nome = "Maria", Idade = 31 },
    new Pessoa { Nome = "Julia", Idade = 25 },
    new Pessoa { Nome = "Laura", Idade = 13 }
};
```

Cada pessoa será referenciada por **p**

A coleção **pessoas** é a origem dos dados

```
var s = from p in pessoas
        where p.Idade > 20
        select p;
```

Projeção (o que será retornado)

Condição para extração dos dados

O retorno de uma expressão LINQ é um objeto `IEnumerable<T>`

O tipo `var` normalmente é utilizado (às vezes ele é obrigatório)

---

---

---

---


---

---

---

---

## Iterando Sobre os Elementos



- A iteração sobre os elementos é feita normalmente com o uso da estrutura *foreach*

```
foreach (var p in s)
{
    Console.WriteLine(p.Nome);
}
```

O compilador infere que **p** é do tipo *Pessoa*

---

---

---

---


---

---

---

---

## Mais um Exemplo do Uso do LINQ



O compilador checka a sintaxe da expressão LINQ

```
var s = from p in pessoas
        where p.Idade > 20 && p.Nome.Contains("a")
        orderby p.Idade ascending
        select p.Nome;
```

Apenas o nome das pessoas é retornado

Mais de uma condição

Ordenação por idade

```
foreach (var nome in s)
{
    Console.WriteLine(nome);
}
```

O compilador infere que **nome** é do tipo *string*

---

---

---

---

---

---

---

---

## Deferred Execution



- Uma expressão LINQ só é executada quando a iteração sobre os elementos é realizada
  - Deferred execution (execução postergada)

```
var s = from p in pessoas
        where p.Idade > 20
        select p;
```



A expressão ainda não foi executada

```
foreach (var p in s)
{
    //...
}
```



Agora a expressão é executada

---

---

---

---

---

---

---

## Immediate Execution



- É possível forçar a execução imediata da expressão através de alguns métodos
  - `ToArray()`, `ToList()`, `ToDictionary()`

```
var s = from p in pessoas
        where p.Idade > 20
        select p;
```

```
List<Pessoa> list = s.ToList();
```

```
Pessoa[] array = s.ToArray();
```

```
Dictionary<string, Pessoa> dict = s.ToDictionary(p => p.Nome);
```

```
List<Pessoa> list = (from p in pessoas
                    where p.Idade > 20
                    select p).ToList();
```

---

---

---

---

---

---

---

## Assemblies & Namespaces



- Para utilizar o LINQ para objetos, o assembly ***System.Core.dll*** deve ser referenciado
  - Projetos do Visual Studio fazem esta referência de forma automática
- Todos os arquivos que usam expressões LINQ devem importar o namespace ***System.Linq***

---

---

---

---

---

---

---

## A Classe *System.Linq.Enumerable*



- Esta classe tem um papel fundamental na estrutura de funcionamento do LINQ
  - Ela adiciona uma série de extension methods à interface *IEnumerable<T>*
- Quando uma expressão LINQ é compilada, ela é transformada em chamadas a estes extension methods

---

---

---

---

---

---

---

## A Classe *System.Linq.Enumerable*



```
var s = from p in pessoas
        where p.Idade > 20
        orderby p.Nome descending
        select p;
```

```
var s = pessoas
        .Where(p => p.Idade > 20)
        .OrderByDescending(p => p.Nome)
        .Select(p => p);
```

O resultado é o mesmo

Métodos como *Where()*, *OrderByDescending()*, *Select()*, etc. são definidos em *Enumerable* como sendo extension methods de *IEnumerable<T>*

---

---

---

---

---

---

---

## O Método *OfType<T>()*



- LINQ não consegue trabalhar com coleções que não usam generics

```
ArrayList pessoas = new ArrayList {
    new Pessoa { Nome = "Maria", Idade = 31 },
    new Pessoa { Nome = "Julia", Idade = 25 },
    new Pessoa { Nome = "Laura", Idade = 13 }
};
```

Erro de compilação

```
var s = from p in pessoas
        where p.Idade > 20
        select p;
```

- A classe *System.Linq.Enumerable* define um método chamado *OfType<T>()*
  - Converte para um tipo genérico

```
var s = from p in pessoas.OfType<Pessoa>()
        where p.Idade > 20
        select p;
```

Transforma um *IEnumerable* em um *IEnumerable<T>*

---

---

---

---

---

---

---

## O Método *OfType<T>()*



- Este método também pode ser usado para filtrar elementos de determinado tipo

```
ArrayList list = new ArrayList { "A", "B", 1, 2 };  
var strings = list.OfType<string>();
```

Retorna apenas os elementos "A" e "B"

## Projetando Tipos de Dados Anônimos



- LINQ permite a criação de novos tipos de dados para o retorno
  - Tipos anônimos aplicados ao *select*

Neste caso, o uso do tipo *var* é obrigatório

```
var s = from p in pessoas  
        where p.Sexo == SexoPessoa.Feminino  
        select new { Nome = p.Nome, Idade = p.Idade };
```

Classe anônima contendo nome e idade das pessoas

```
foreach (var p in s)  
{  
    Console.WriteLine(p.Nome + " -> " + p.Idade);  
}
```

## Definindo Variáveis



- LINQ permite a criação de variáveis, a fim de evitar a repetição do mesmo código
  - Uso da instrução *let*

```
var s = from p in pessoas  
        let i = p.Idade;  
        where i > 20  
        select new { Nome = p.Nome, Idade = i };
```

Cria a variável *i*



## Operações de Agregação



- LINQ possui operações que agregam os elementos da coleção
- Só podem ser chamados como métodos
  - Extension methods de *IEnumerable<T>*

```
double[] notas = { 9.5, 7.0, 6.5, 8.0 };
```

<code>int count = notas.Count();</code>	Quantidade
<code>double min = notas.Min();</code>	Mínimo
<code>double max = notas.Max();</code>	Máximo
<code>double avg = notas.Average();</code>	Média
<code>double sum = notas.Sum();</code>	Soma

---

---

---

---

---

---

---

## Operações com Conjuntos



- LINQ define alguns extension methods para operações com conjuntos
  - União, interseção, subtração e concatenação

```
char[] letras1 = { 'A', 'B', 'D', 'F' };  
char[] letras2 = { 'A', 'C', 'D' };
```

<code>var union = letras1.Union(letras2);</code>	A, B, D, F, C
<code>var inters = letras1.Intersect(letras2);</code>	A, D
<code>var subtr = letras1.Except(letras2);</code>	B, F
<code>var subtr = letras1.Concat(letras2);</code>	A, B, D, F, A, C, D

O método *Distinct()* remove elementos duplicados do conjunto

---

---

---

---

---

---

---

## Cláusula *from* Composta



- O uso do *from* composto é útil quando é preciso acessar mais de um nível de objeto

```
class Pessoa  
{  
    public string Nome { get; set; }  
    public int Idade { get; set; }  
    public List<Brinquedo> Brinquedos { get; set; }  
}  
  
class Brinquedo  
{  
    public string Nome { get; set; }  
}
```

```
var s = from p in pessoas  
        from b in p.Brinquedos  
        where p.Idade < 10 && b.Nome == "Boneca"  
        select p;
```

---

---

---

---

---

---

---

## Agrupamento de Dados



- LINQ permite agrupar dados com base em uma chave

```
var s = from p in pessoas
        group p by p.Idade into g
        select new { Idade = g.Key, Num = g.Count() };
```

Cria um grupo *g* onde a idade da pessoa é usada como chave

O retorno é de um tipo anônimo que contém a idade e a contagem

- É possível agrupar usando outros critérios

```
var s = from p in pessoas
        group p by p.Idade into g
        select new { Idade = g.Key, NumBrq = g.Avg(p => p.NumIrmaos) };
```

Agrupar por idade e mostra a média do número de irmãos

## O Operador *join*



- O *join* permite combinar múltiplas fontes de dados para extrair informações

```
List<Pessoa> pessoas = new List<Pessoa>()
{
    new Pessoa { Nome = "Maria", Idade = 13 },
    new Pessoa { Nome = "Julia", Idade = 9 },
    new Pessoa { Nome = "Laura", Idade = 13 }
};
```

```
List<Aluno> alunos = new List<Aluno>()
{
    new Aluno { Nome = "Maria", Nota = 8.5 },
    new Aluno { Nome = "Julia", Nota = 9.0 },
    new Aluno { Nome = "Laura", Nota = 7.0 },
};
```

O nome é usado para combinar as coleções

```
var s = from p in pessoas
        join a in alunos on p.Nome equals a.Nome
        select new { Nome = p.Nome, Idade = p.Idade, Nota = a.Nota };
```

## O Operador *join*



- O método *DefaultIfEmpty()* pode ser usado para trazer dados que existem em apenas uma das coleções

```
List<Pessoa> pessoas = new List<Pessoa>()
{
    new Pessoa { Nome = "Maria", Idade = 13 },
    new Pessoa { Nome = "Julia", Idade = 9 },
    new Pessoa { Nome = "Laura", Idade = 13 },
    new Pessoa { Nome = "Ines", Idade = 12 }
};
```

```
List<Aluno> alunos = new List<Aluno>()
{
    new Aluno { Nome = "Maria", Nota = 8.5 },
    new Aluno { Nome = "Julia", Nota = 9.0 },
    new Aluno { Nome = "Laura", Nota = 7.0 },
};
```

```
var s = from p in pessoas
        join a in alunos on p.Nome equals a.Nome into pa
        from a in pa.DefaultIfEmpty()
        select new { Nome = p.Nome, Idade = p.Idade, Nota = a == null ? 0 : a.Nota };
```

## Geração de Dados



- LINQ também possui métodos para gerar dados
  - Métodos estáticos na classe *Enumerable*
    - *Range()*
    - *Empty()*
    - *Repeat()*

---

---

---

---

---

---

---

## Geração de Dados – *Range()*



- Gera um intervalo numérico
  - Apenas números inteiros

```
var v = Enumerable.Range(1, 5);
```

1, 2, 3, 4, 5

- O método *Select()* pode ser usado para modificar o comportamento do intervalo

```
var v = Enumerable.Range(1, 5).Select(n => n * 2);
```

2, 4, 6, 8, 10

---

---

---

---

---

---

---

## Geração de Dados – *Empty()*



- Gera uma coleção vazia
  - Útil em situações onde é necessário fornecer como parâmetro ou retornar uma coleção vazia

```
var v = Enumerable.Empty<int>();
```

*IEnumerable<int>*

---

---

---

---

---

---

---

## Geração de Dados – Repeat()



- Gera uma coleção com o mesmo elemento repetidas vezes

```
var v = Enumerable.Repeat("A", 4);
```

A, A, A, A

---

---

---

---

---

---

---



Softblue

---

---

---

---

---

---

---